15-462 Project 3: Ray Tracing

Release Date: Thursday, March 12, 2009

Due Date: Thursday, April 9, 2009 at 23:59:59

Starter Code: `http://www.cs.cmu.edu/~15462/proj/03/p3.tar.gz`

1 Overview

In the first two projects, you learned how to use OpenGL and GLSL to render scenes. For this project, we will be moving away from OpenGL (finally!) and asking you to implement a basic ray tracer that can handle shadows, reflections and refractions. All rendering will be done with software, only using OpenGL to display the final image to the screen (we provide that part for you). You will then use this as a jumping point in project 4 to implement an even more advanced rendering model.

As a warning, this assignment is far more code intensive than either of the previous two. However, it should be more straightforward since there is not any OpenGL involved and since the textbook is a very excellent resource for this topic. So **start early**. *Do not* wait until the last week to start. There is a lot of code, and the fourth project is dependent on you correctly implementing nearly all of the minimum requirements for project 3.

Before you start, you will need to download the staff code from the course website and update your code according to section 2 of this document.

Chapter 10 of the Shirley textbook will be the most useful resource for this assignment, so we strongly recommend that you look at it before starting this assignment. Nearly all topics covered in this handout are also covered in the textbook (though sometimes we present slightly simplified mathematics). Any references to the Shirley text, unless noted otherwise, are in chapter 10.

The slides from Lectures 12, 13, and 14 of class also contain material that you may find useful in writing your ray tracer.

1.1 Submission Process

Your handin directory may be found at /afs/cs.cmu.edu/academic/class/15462-s09-users/andrewid/p3/ You should submit the entire contents of the handout, your modified files, and any screen shots in this directory.

To test your code, we will simply `cd` into the directory, then run `make clean; make`. Therefore, your code should should be placed in the root directory uncompressed (i.e. the makefile's absolute path should be

`.../15462-s09-users/`*andrewid*`/p3/Makefile`.). **DO NOT** submit a compressed file or binary files. So, if you made it only windows, delete the `Debug` and `Release` folders, and if you made it on a Unix system, run `make clean` before submitting.

In addition, you must fill out the p3.txt file in the handout, describing which features you have implemented for project 3, and which you have not. In particular, if you have missing features, be sure to explain how much you accomplished and why. If you did extra work, be sure to explain what that was and how you did it. Furthermore, you should detail anything you think we need to know to understand your code.

Regardless of what machine you use to code your assignment, it must compile and run correctly on the machines in the Wean 5336 cluster. There are a lot of students in the course and only 25 machines, so you probably don't want to wait until the last minute to test it, even more so because you will want a substantial amount of time to render your scenes.

For late submissions once the handin directories have been locked, please make sure that you zip or tar your files and email them to ALL of the staff. Also, please make sure that you remove all Debug/Release folders or extraneous files that can bloat your file size (so that you can actually email them and so that the staff can download them quickly).

2 Handout/Requirements

2.1 Required Tasks

- Implement `rt_raytrace` as defined by the spec to ray trace scenes.
- Write intersection tests for the three geometric objects in the scene.
- Implement the basic ray tracing algorithm by sending a ray from the eye to all objects in the scene.
- Add direct illumination and shadows by sending shadow rays to the lights.
- Add specular reflections by sending reflected rays into the scene.
- Add refractions by sending transmission rays through dielectric materials.

2.2 Extra Credit

See section 7.1 for information about possible extra-credit ideas.

2.3 Files to Update/Edit

The handout for project 3 includes the following files:

- `staffldr.cpp`
- `app.cpp`
- `project.h`
- `raytrace.cpp`

- `vec/mat.cpp/.h`
- `vec/quat.cpp/.h`

The only required file is `staffldr.cpp`, which contains the staff scenes. All other files contain either starter/helper code or bug fixes, which you may use if you wish.

We provide you with a bit of starter code for the `rt_raytrace` function. See section 4.2.3 for details on what we give you and what you need to do with it.

2.3.1 Note About Old Project Code

We do not require that your submission support the requirements from project 1 and 2. Therefore, you can consider removing most of the extraneous code from your project, such as the shaders. We will not be grading these, and removing them will simplify your code base and make things more maintainable.

3 Ray Casting and Intersection Tests

3.1 Ray Casting

The primary ability needed by the ray tracer is the ray cast function, which sends out a given ray $p(t) = e + dt$ into the scene and returns the first object intersected by the ray and the time at which the intersection occurs. This basic function will be used by all other parts of the ray tracer to perform such tasks as casting eye rays, shadow rays, reflected rays, and transmission rays.

The ray casting function also takes a bound on the time—a minimum and maximum time in which to search for intersections. Only intersections that occur within the bound are considered. So, for example, obviously the minimum bound is at least 0 since you do not wish to consider intersections that occur behind the ray's starting point. The exact bound is different for each type of ray, but generally, for eye rays it corresponds to the near and far planes, and for other rays it runs from $[\epsilon, \infty)$ where ϵ is a very small positive value to prevent the ray from intersecting the object from which it was cast.

You must write intersection tests for each of the objects. You may wish to write this code in conjunction with the basic ray tracing algorithm outlined in the next section so that you can test your intersection tests along the way.

You may wish to declare a pure virtual function in the `Geometry` class and implement it in each derived class in order to perform intersection tests. In general, we suggest you heavily modify the geometry classes to assist with ray tracing. You may make any changes you wish, as long as you do not change the constructor signatures.

3.2 Sphere-Ray Intersection

A sphere can be defined by its center $c = (x_c, y_c, z_c)$ and its radius r. It can be represented implicitly the following equation:

$$(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 - r^2 = 0$$

Written in vector form:

$$|p - c|^2 - r^2 = 0$$

Plugging in $p(t)$ for p, we simply solve for t to find the time of the intersection. The complete derivation is available in the text. Basically, we must solve a quadratic equation, which means there are either 0, 1, or 2 solutions. These correspond to no intersection, glancing off the sphere, or going through the middle of the sphere, respectively. The time of intersection would then be the minimum (if any) of the solutions.

3.3 Triangle-Ray Intersection

The method that we recommend for solving triangle-ray intersection is the same method that is explained in the Shirley text. This method uses barycentric coordinates. You will need these later to compute the color, so we suggest you store them after solving for later use.

3.4 Water Surface-Ray Intersection

If you constructed your water surface as instructed, then performing an intersection test reduces to that of performing an intersection test on a generic triangle mesh. The simplest method is to perform an intersect test on each triangle mesh and, if it intersects any, return the first triangle intersected.

Of course, that is a tremendously slow method, and so it would be much better to have a sub-linear method that involved some kind of spatial optimization. Traditional ones like octrees or BSP trees work, but another possibility is to abuse our knowledge that the mesh is a heightmap to come up with an even better one. Consider the invariants we have: the mesh is of uniform density along both the x and z axes, and is far greater in dimension along those axes than along the y axis. What possible optimizations can be done? As a hint, consider projecting the water surface's mesh into only 2 dimensions and performing a broad-phase check in 2D.

3.5 Optimizations

Each of the above describe a narrow-phase intersection test with a ray. However, some of them (particularly the water surface test) are rather computationally expensive. We can mitigate this by providing a less accurate but much cheaper broad-phase intersection test. You would first run the broad-phase check, which would narrow down the list of objects on which you would have to perform the narrow-phase check, and then you can do far fewer narrow-phase checks for each ray.

One simple broad-phase optimization is to surround each object with a bounding box or sphere. You would first check for intersection with the box or sphere, and if this succeeds, you would then perform the narrow-phase check. Ray-box and ray-sphere intersection tests are also in the text.

A more sophisticated broad-phase is using spatial data structures to achieve sub-linear performance. Similar to many search algorithms, ray casting can be optimized by a “divide and conquer” approach, whereby we create a suitable data structure to store the geometries in the scene and to optimize searching for intersections. Such data structures include octrees, BSP trees, and uniform spatial subdivision.

We highly suggest you at least implement bounding boxes. You can likely get away without any further search optimizations for this project, though you will gain experience with spatial data structures on the next project.

4 Basic Ray Tracing and Eye Rays

4.1 The Ray Tracing Function

Now that you have methods to intersect objects, you can use these methods to begin building the basic ray tracing algorithm. We use our ray casting function to create the basic recursive ray tracing function that, given a ray $p(t) = e + dt$, returns the color of that ray. This will be used by eye rays, reflected rays, and transmission rays.

Basically, the ray trace function invokes ray cast to determine if an object is hit within the time bounds. If so, it computes the color on the object at that point. Otherwise, it returns the color of the background. In our case, since we are using a sphere map, you should look up the background color with said sphere map.

4.2 Eye Rays

You can use the ray tracing function to create the basics of your ray tracer. The idea is simple: for every pixel on the screen, you will want to compute the “eye ray” coming out of that pixel and cast it into the scene. If a ray intersects an object, you will want to return the color of the pixel at that point of intersection.

At first, you probably want to make a very simply color computation. For example, return only the diffuse color at that point, or perhaps return the color as computed by Phong illumination. Of course, the actual computation is much more involved (see section 6), but this will allow you to test you eye rays.

4.2.1 Computing the Image Plane

Before we can cast eye rays, we must first construct the image plane, which will then give us the position of each “pixel” in world space. The book has slightly different derivation, but here we present one in which you first compute the image plane of the camera, then use it to compute each eye ray.

The image plane can be constructed using the data in the `Camera` class. Here, we let e_c be the position, d_c be the direction u_c be the up vector, n be the near clipping distance, θ be the field of view, and r be the aspect ratio.

The image plane is the plane whose normal is d_c, and it is located n distance away from the camera position. So a point on the plane (corresponding to the center of the screen) is $e_c + nd_c$.

We also need the axes of the image plane, I_x and I_y, which correspond to the two major axes of the screen. The up vector defines I_y (so $I_y = u_c$) and we can find I_x using the up and direction vectors ($I_x = d_c \times u_c$).

Lastly, we need to compute the dimensions of the image plane using the field of view and aspect ratio. Using basic trigonometry, we compute the half-height, h, of the image plane with $h = 2n\tan(\theta/2)$. The half-width w is simply then $w = hr$.

4.2.2 Computing the Eye Rays

We aim to compute $p(t) = e + dt$ as our eye ray. We already have e: it is simply the position of the camera. Computing d may seem complicated, but, given our image plane, is actually quite simple. We can generalize a pixel's position on the plane as a vector $v \in [-1,-1] \times [1,1]$ (a 2D vector whose components range from -1 to 1. $(-1,-1)$ is the bottom left of the plane, and $(1,1)$ is the top right of the plane. This value can be computed for each pixel given the pixel's index and the total number of pixels in each direction. We can compute the point on the plane through which the eye ray passes, then subtract from e to get d.

So d is the center of the image plane plus the x component times the image plane's x axis plus the y component times the image plane's y axis minus the camera's position. Using the variables as defined in section 4.2.1, we have

$$\begin{aligned} d((x,y)) &= (e_c + nd_c) + (I_x wx/2 + I_y hy/2) - e_c \\ &= n(d_c + \tan(\theta/2)(rx(d_c \times u_c) + yu_c)). \end{aligned}$$

4.2.3 The `rt_trace_pixel` Function

The starter code provides you with an implementation of `rt_raytrace` that iterates over each pixel for you. It will invoke the unimplemented `rt_trace_pixel` function to get the color at each pixel. If you wish to use our starter code, you should implement this function. However, if you wish, you may implement `rt_raytrace` in another manner, as long as it meets the function specification described in `raytrace.cpp`.

The basic ray tracing algorithm you will have written so far simply returns the color of the first object that it intersects. If your intersection tests are correct, then your code should currently return a scene with no shading and only flat colors.

5 Computing the Recursive Rays

The next step is to compute the color correctly. However, this requires your ray tracer to be able to correctly send out the remaining 3 types of rays: shadow, reflected, and transmission.

Note that all of these are covered extensively in Shirley, with full derivations for the math involved. You should consult the text for more detail.

5.1 Using the Ray Cast Function

Each of the recursive rays will also use the ray cast functionality. However, unlike eye rays, which are fired from the camera, all of these rays are fired from the point of intersection p with an object in the scene. This point is given to us by the eye ray's intersection tests, and so all we need to do is compute the direction of the new recursive ray.

5.1.1 Recursion Depth

Two of these rays will be used in recursive calls to you ray tracing function. However, this leaves open the possibility for infinite recursion, as rays bounce and refract around the scene forever. The ray tracer must be stopped somewhere. You want this to happen once the contribution has become small, so it is not noticeable.

There are a few ways to accomplish this, but one simple way is to simply cap the maximum recursion depth of the ray tracer. Once the max depth is reached, reflection and refraction are not considered. We require your ray tracer to support up to at least a depth of 3.

5.1.2 Slop Factor

The other major issue with recursive ray tracing is the fact that the ray's origin is on the surface of a geometry. This means that the intersection test will likely return $t = 0$, since the ray is colliding with an object at time 0.

One easy way to correct for this is to introduce a slop factor $\epsilon > 0$ as the minimum time bound, to prevent the collision with the ray origin from occurring. ϵ should be a very small positive number.

5.2 Shadow Rays

Shadow rays are rays that are cast from the intersection point to a light source to determine the visibility of that light source. When computing direct illumination (see section 6.2.1), one must determine whether a light source is even visible from the intersection point. For this we can simply use our ray cast function to determine if there is another object in the way. If a shadow ray hits any object, then there is no contribution from that light.

Note: This actually breaks down in the face of refraction, since a transparent object doesn't actually block light rays from reaching a point It in fact can concentrate them more, as we will explore in project 4. For project 3, you can simply ignore dielectrics when casting a shadow ray. That is, only opaque objects (see 5.4 for information on which objects are opaque) can cause shadows. Pretend dielectrics aren't even part of the scene for shadow rays. This is, of course, physically inaccurate, but we prefer this simple model to the more accurate one.

5.3 Reflected Rays

Reflected rays are computed exactly the same as done by your code from project 2. We take the incoming ray, bounce it off the normal, and invoke the ray tracer on this reflected ray. Given the incoming ray V and the normal N, the reflected ray R is

$$R = V - 2(N \cdot V)N.$$

5.4 Transmission Rays

Certain materials allow the transport of light through them. These materials are known as dielectrics and allow for the refraction of light. In our scene, dielectrics are represented using the `Material::refraction_index` attribute. We use 0 as a special case for opaque objects, and any non-zero value to represent the refraction index of that material. We provide a small helper routine `Material::is_opaque` that returns `false` if a material is a dielectric.

Recall that Snell's law gives us the relation

$$n \sin(\theta) = n_t \sin(\phi)$$

between the angles of incidence and the refractive indices of two different dielectrics. We can use this fact to compute the direction of the transmission ray. The full derivation is on page 213 of Shirley.

We replicate the result here. n is the index of refraction of the old dielectric, n_t the index of the new, D the incoming ray, and N the normal. The refracted direction T is

$$T = \frac{n(D - N(D \cdot N))}{n_t} - N\sqrt{1 - \frac{n^2(1 - (D \cdot N)^2)}{n_t^2}}.$$

Note that if the result is imaginary, then there is no refraction. It is total internal reflection.

5.4.1 Tracking the Current Refraction Index

Tracing a "pool" scene is a bit tricky since we must track the current refraction index so we know which values to put into the equation. We assume that the ray trace starts in the scene's background refraction index, which is given by the `Scene` class. From there, any time you enter a dielectric, the current index changes. Once you leave, the index goes back to what it was before.

The staff suggest using a small stack to track this information. You can determine whether you're entering or leaving a dielectric based on the direction of the normal vector. The normal points out, so if the dot product of the normal and the incoming ray is negative, the ray is entering. Otherwise, the ray is exiting. Be careful, since rays can reflect in between refractions (via total internal reflection, etc.).

One other small issue to consider is that of floating point error. It may be the case that your ray casting, for reasons caused by errors inherent in floating point, missed an entrance/exit from a dielectric. This can cause your stack to become corrupt/invalid. It may be impossible to avoid this, so your best bet is to have code to handle the case where the stack becomes invalid.

6 Computing the Color

Once we have determined when and where an intersection occurs, we must compute the color at that point. Your code must utilize the recursive ray tracing calls as described in section 5.

6.1 Computing the Needed Values

First you must determine a few things at the point p. Of chief interest are the material, the normal N, the texture coordinates (u, v), the viewing ray V, and each light ray L. V and L can be easily computed. The others may be computable directly (as in the case of a sphere) but may need to be interpolated.

6.1.1 Interpolation

For triangles, the way to compute the values at a given point is by interpolation. This requires the barycentric coordinates α, β, γ computed in the intersection test. To get the value of a vector, color, or float at any given point $p = \alpha a, \beta b, \gamma c$ where a, b, c are the vertices of the triangle, we simply interpolate the value. So, for example, to compute the diffuse color k_d at p, where c_i is the diffuse color at vertex i, we have

$$k_d = \alpha c_a + \beta c_b + \gamma c_c.$$

This computation works identically for all vectors, floats, or colors. So we can interpolate the normal, the texture coordinates, and every value of the material.

Note that since the water surface has a uniform material, we only really need to interpolate the normal and texture coordinates. However, the `Triangle` class has a different material on each vertex, and so you must interpolate all the values of the material to get the correct effect. In previous projects you ignored these per-vertex materials due to OpenGL constraints, but since we are doing all the rendering ourselves, you can have a different material for each vertex.

6.2 The Three Components

We require that your ray tracer support direct illumination, specular reflection, and refraction. Note that the color computations vary based on the type of object. In our simple model, we support only fully opaque objects and fully transparent objects. In the former, only direct illumination and specular reflection contribute to the color. In the latter, only specular reflection and refraction contribute to the color. The exact ways in which these are computed and combined are described in this section.

6.2.1 Direct Illumination

Your ray tracer should support the basic Phong illumination model for its direct illumination that we have been using for the past two projects. However, we do not need to use the Phong specular component since we have a more accurate specular computation. You may still include a Phong specular component if you wish, since you specular reflection computations do not support semi-glossy surfaces well, but it is not required. Therefore, direct illumination consists of ambient and diffuse colors.

Ambient, as always, is the ambient color of light, c_a, multiplied by the material's ambient color, k_a. The ambient light color c_a is given by `Scene::ambient_light`.

Diffuse is computed for each light i in the set of lights I. We multiply the color of the light c_i by the diffuse material k_d and the dot product of the normal N and the light vector L. However, we must first use a shadow ray to determine whether the light actually contributes at that point. If the shadow ray hits an object between the point and the light, then there is no contribution from that light.

This is also the place where the object's texture comes into play. The entire direct illumination component should be multiplied by the texture at that point. We provide a texture lookup function for you (`Material::get_texture_color`). You must provide the texture coordinates, whose computation is described in 6.1.

So, all together, the color at a point p is

$$c_p = t_p(c_a k_a + \sum_{i \in I} b_i c_i k_d \max\{N \cdot L, 0\}).$$

where b_i is 0 if the shadow ray from p to i intersects an object, 1 otherwise.

6.2.2 Reflection and Refraction

The color contributions from specular reflection and refraction are from recursive calls to the ray trace function. Using the computed reflection/transmission rays, you compute the color of that ray. In the case of refraction, since you need to support only fully clear transparent objects, that is the contribution. In the case of reflection, you must multiply the returned color by the material's specular color, which is given by `Material::specular`.

6.3 Putting It All Together

For opaque surfaces, we simply sum the two components. You compute the direct illumination and specular terms, then sum them to get the final color. The story is a little more complex for dielectrics.

6.3.1 The Fresnel Effect

For dielectrics, we must consider the Fresnel equations, which describe how much light reflects and how much refracts on a given surface. You used these equations (or rather, an approximation of them) in project 2, but we ignored the refraction component and simply used a solid color. Now that we have the ability to do both refraction and true specular reflection, we can use both components to create a much more realistic effect.

As before, the Schlick approximation of the Fresnel effect is described on page 214 of Shirley. You should compute the Fresnel coefficient R. Given that and the values of specular reflection c_r and refraction c_f, the final color is

$$c_p = Rc_r + (1 - R)c_f.$$

Note: if there was no refraction component (due to total internal reflection), then just use $R = 1$.

7 Before You Start/Words of Advice

Writing a ray tracer is a substantial undertaking, which is why we have allotted more time for it than the first two projects. We are giving you three weeks, plus spring break, to complete this assignment, so you will want to take advantage of this. There is a lot of code to write, a lot of math to think through, and a lot of time needed to render your scenes, so you will **not** want to wait until a week before the assignment is due to start.

Also, rendering scenes with your ray tracer is nearly impossible in real time and can take anywhere from several seconds to several hours depending on your implementation. You will want to set aside at least a day or two, if not more (especially if you make your ray tracer distributed) just for rendering. Remember that other people will need to use the cluster, so we strongly advise you to not render your scenes on the 5336 machines when other people need to work locally. Please schedule your rendering time carefully and do not monopolize the machines.

A final thing to take into consideration is that you will have to use your ray tracer as a starting point for the final project. You will want to make sure that you have a thorough understanding of how your code works and keep it clean and commented in order to make writing project 4 much easier.

7.1 Extra Credit

Any improvements/optimizations to the ray tracer above the minimum requirements can be cause for extra credit. Some possibilities are:

- Make your ray tracer distributed by adding any number of the following:
  - Anti-aliasing
  - Soft shadows
  - Depth of field
  - Glossy reflection
  - Motion blur
- Speed up the performance of your ray tracer by adding one of the following:
  - Bounding boxes/bounding spheres
  - A spatial data structure (we suggest an octree)
- Ray trace your own geometry by writing an intersect test for it and building a scene demonstrating it.

As with previous projects, extra credit will be considered separately from the main assignment. We are still considered how to compute points, but rest assured you will be getting credit for the extra work you do (or have already done on earlier projects).